Midterm 1: Compsci 201

Tabitha Peck

February 15, 2013

Name: _____

NetID/Login: _____

Honor code acknowledgment (signature) _____

This test has 14 pages (with a help page at the end), be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting function or method names exactly right. Assume that all libraries and packages are imported in any code you write.

	value	grade
Problem 1	15 pts.	
Problem 2	12 pts.	
Problem 3	12 pts.	
Problem 4	8 pts.	
Problem 5	12 pts.	
Problem 6	16 pts.	
TOTAL:	75 pts.	

PROBLEM 1 : (*Book Shelf: 15 points*)

You have a collections of books to put on your bookshelf. You have decided to order them largest to smallest by number of pages and to break ties alphabetically by title. The method `sortBooks` will help you, but you need to complete the implementation of the `Book` class for sortBooks to work. The method `sortBooks` has two parameters, `bookTitle` and `pageCount` which hold book titles and page counts respectively, such that the title in `bookTitle[i]` has `pageCount[i]` pages.

Example:

Input:

`bookTitle` = [“Catcher in the Rye”, “Adventures of Huck Finn”, “Lord of the Flies”, “Sun Also Rises”]

`pageCount` = [253, 312, 198, 253]

Output:

“Adventures of Huck Finn” “Catcher in the Rye” “Sun Also Rises” “Lord of the Flies”

Code: sortBooks

```
public void sortBooks(String[] bookTitle, int[] pageCount){
    ArrayList<Book> list = new ArrayList<Book>();
    for(int i = 0; i < bookTitle.length; i++)
        list.add(new Book(bookTitle[i], pageCount[i]));
    Collections.sort(list);
    for(Book b: list)
        System.out.println(b.myTitle);
}
```

Instance variables and constructor: 4 points

Complete the implementation of class `Book` by adding instance variables and completing the constructor so that the class `Book` will work with the code given above.

```
public class Book implements Comparable<Book>{
    \\ add your instance variable(s) here

    private String myTitle;
    private int myPages;

    \\ complete the constructor definition and don't forget to define your parameters

    public Book(String title, int pages){
        myTitle = title;
        myPages = pages;
    }
```

Question continues on the next page.

Equals and hash code: 6 points

Complete the equals and hashCode methods below.

```
public boolean equals(Object obj){
    if (obj == this) {
        return true;
    }
    if (obj == null || obj.getClass() != this.getClass()) {
        return false;
    }

    Book temp = (Book) obj;



    if(this.myTitle.equals(temp.myTitle) && this.myPages==temp.myPages)
        return true;
    return false;

}


public int hashCode(){



    return (myTitle + myPages).hashCode();



}
```

compareTo: 5 points

Complete the compareTo method below.

```
public int compareTo(Book arg0) {



     int pages = arg0.myPages - this.myPages;
    if(pages == 0)
        return this.myTitle.compareTo(arg0.myTitle);
    return pages;




}
```

PROBLEM 2 : (*Big-Oh: 12 points*)

For each of the following, give the running time in terms of the parameter `n` in big-Oh notation. You must justify your answer. (2 points for each running time, and 1 point for each justification)

```
public int numberOne(int n){
    int answer = 1;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n*n; j++)
            answer++;
    return answer;
}
```

Big-Oh: $O(N^3)$ Justification: j-loop is N^2 since j increments by 1 to N^2. Out loop executes n times, each one is n^2, that's n^3

```
public int numberTwo(int n){
    int answer = 1;
    for(int i = 0; i < n; i++)
        answer++;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            answer++;
    return answer;
}
```

Big-Oh: $O(N^2)$ Justification: the first i-loop is $O(n)$ The second i-loop executes n times, each iteration is $O(n)$ because j-loop is O(n), so this is $O(n^2)$. Total is $O(n^2) + O(n) = O(n^2)$

```
public int numberThree(int n){
    int answer = 1;
    for(int i = 1; i <= n; i=i*2)
        answer++;
    return answer;
}
```

Big-Oh: $O(lgN)$ Justification: i doubles until it reaches n, e.g., 10 times to get to 1024, that's log

---

```
public int numberFour(int n){
    int answer = 1;
    for(int i = 1; i <= n; i++)
        for(int j = 0; j < i; j++)
            answer++;
    return answer;
}
```

Big-Oh: $O(N^2)$ Justification: inner loop will run 1 time, then 2, the 3, etc. $\sum_{i=1}^{n} i = \frac{n^2+n}{2} = O(n^2)$

PROBLEM 3 : (*Lists: 12 points*)

Part A: 4 points

What is the difference between an ArrayList and a LinkedList?

An ArrayList is implemented using an array and a linkedList is implemented as a linked list with nodes. An arrayList is very fast at getting elements from a specific index, while linkedLists are much faster at adding and removing elements from the beginning or middle of the list. What do an ArrayList and a LinkedList have in common?

ArrayList and LinkedList are both lists–an ordered collection of items. They both use the List interface in Java and can call the same list methods, such as .add.

Part B: 4 points

The following code removes all elements from the list passed as a parameter:

```
public void removeAll(List<String> list){
    int theSize = list.size();
    for(int i = 0; i < theSize; i++)
        list.remove(0);
}
```

What is the running time of the call `removeAll(list1)` of n-element `list1` using the declaration

```
List<String> list1 = new ArrayList<String>();
```

Explain your answer.

remove from the beginning of an ArrayList is O(N), this is called N times. $O(N^2)$

Question continues on the next page.

What is the running time of the call `removeAll(list2)` of n-element `list2` using the declaration

```
List<String> list2 = new LinkedList<String>();
```

Explain your answer.

remove from the beginning of a linkedList is O(1), this is called N times. $O(N)$

Part C: 4 points

The method `remove`, shown below, has the same running time as `removeAll` but does not perform in the same way.

```
public void remove(List<String> list){
    for(int i = 0; i < list.size(); i++)
        list.remove(0);
}
```

Explain why `remove` and `removeAll` have the same running time.

The loop will run $N/2$ times. Since big-oh removes constants, the loop runs $O(N)$ times, which is the same as the previous loop.

If `list1` and `list2` each contain n-elements, how many elements will `list1` and `list2` have after calling `remove(list1)` and `removeAll(list2)`?

list1 will have N/2 elements

list2 will have zero elements

PROBLEM 4 : (*Hashing: 8 points*)

Part A: 4 points

The following numbers are inserted into a 10 element hash table (indexed 0 - 9) in the order shown

[2491, 5749, 3584, 1257, 5981, 3475, 5299, 5429, 5241]

Using the hash function $h(x) = x \ mod \ 10$ draw the resulting hash table using separate chaining (i.e. linked-lists).

0 ——

2491 → 5981 → 5241

1 ——

2 ——

3584

3475

3 ——

1257

4 ——

5749 → 5299 → 5429

Part B: 4 points

Adding elements to a hash table is often very fast. However, not always. Explain how insertion into a hash table can become slow. Your discussion should talk about hashCode.

HashCodes that map different objects to the same key, and hash functions that map multiple objects to the same location in a hash table will cause collisions. When collisions happen the time to insert and remove from the hashMap increases. In the worst case, if all objects map to the same location, you essentially have a linked list, and the time to add and remove will be the length of the list, $O(N)$, where if you have no collisions add and get are O(1).

PROBLEM 5 : (*Fruit and Veg: 12 points*)

Your friends decided to make a game of who can eat more fruits and vegetables to encourage healthy eating since everyone says to eat more fruits and vegetables. They determined that each player's score for the week would be:

`(number of times most common food is eaten) * (total number of unique foods eaten)`

That is, if you ate `"banana orange apple lettuce avocado banana apple banana"` you ate bananas 3 times and you ate 5 different foods (banana orange apple lettuce avocado). Therefore, you would get (3 * 5) = 15 points.

You decided to write a program to help your friends determine the winner.

Part A: 4 points

Complete the helper method `buildMap` which takes as a parameter a space-separated list of food someone ate, and returns a HashMap with `key`–food item (as a String), and `value`–number of times the food item was eaten.

```
public HashMap<String, Integer> buildMap(String str){

        HashMap<String, Integer> map = new HashMap<String, Integer>();
        String[] f = str.split(" ");
        for(String s: f){
            if(map.containsKey(s)){
                map.put(s, map.get(s) + 1);
            }
            else{
                map.put(s, 1);
            }
        }
        return map;
```

Question continues on the next page.

Part B: 4 points

Complete the helper method `calculateScore` which takes as a parameter the HashMap built in `buildMap` and returns the player's score as defined on the previous page.

```
public int calculateScore(HashMap<String, Integer> map){


        int max = 0;
        for(String s: map.keySet()){
            if(map.get(s) > max)
                max = map.get(s);
        }
        return map.size() * max;
```

Part C: 4 points

Complete the method `whoWon` that takes parameters `food`–a String array of a space-separated lists of the food each person ate for the week, and `name`–a String array of the names of each friend who is playing, such that `name[i]` ate `food[i]`. The method `whoWon` should return the name of the person with the most points. Do not worry about breaking ties.

```
public String whoWon(String[] food, String[] name){

        String winner = name[0];
        int score = 0;

        for(int i = 0; i < food.length; i++){
            HashMap<String, Integer> foodMap = buildMap(food[i]);

            int aScore = calculateScore(foodMap);

            if(aScore > score){
                score = aScore;
                winner = name[i];
            }
        }
        return winner;
```

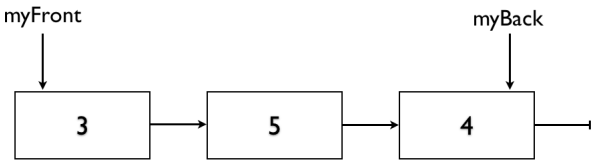
PROBLEM 6 : (*Linked lists: 16 points*)

In class we used a linked-list to implement a stack. Linked lists are commonly used to implement a queue, which is how it is done in Java. For this question you will implement an integer queue using linked lists. Your queue will use the `Node` class below.

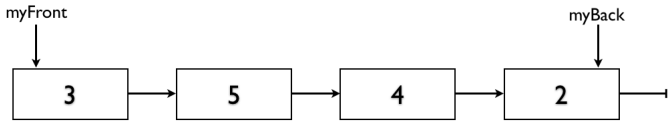
```
public class Node{
    Node myNext;
    int myData;

    public Node(int data, Node next){
        myNext = next;
        myData = data;
    }
}
```

Part A: 4 points

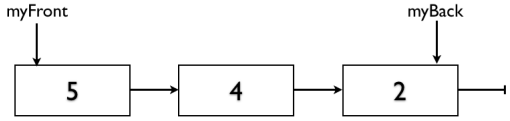
The queue below stores the values (3,5,4) where 3 was added first to the queue, then 5, and then 4. The node `myFront` points to the front of the queue (from which elements are dequeued) and `myBack` points to the back of the queue (to which elements are added).



Draw the resulting queue after an `enqueue` of the value 2. That is, you will add a new node to your queue that holds the value, 2. Note: `myBack.myNext == null`.



After an `enqueue` of 2, draw the resulting queue after a `dequeue` is called.



Part B: 4 points

Below is a partially completed integer queue with both front and back pointers, `myFront` and `myBack`. The number of elements in the queue is held in `mySize`. This code uses the Node class on the previous page.

```
public class intQueue {
    private Node myFront;
    private Node myBack;
    private int mySize;
```

Complete `dequeue`. You can assume `dequeue` isn't called unless there is at least one element in the queue.

```
//remove a node from the beginning of the linked list
    public int dequeue(){


        int data = myFront.myData;
        myFront = myFront.myNext;
        mySize--;
        return data;


    }
```

Part C: 4 points

The code in the `if` handles an empty queue condition. Complete `enqueue` by adding code to the `else`, when the queue is not empty.

```
//add a node to the end of the linked list
        public void enqueue(int data){
        if(myBack == null){  //the queue is empty
            myBack = new Node(data, null);
            myFront = myBack;
            mySize++;
        }
        else{ //the queue is not empty
            Node newBack = new Node(data, null);
            myBack.myNext = newBack;
            myBack = newBack;
            mySize++;
        }
    }
```

Part D: 4 points

What is the running time (big-Oh) of the `enqueue` and `dequeue` methods?

`enqueue`: O(1)

`dequeue`: O(1)

The implementation of `intQueue` used both a front and back pointer. What would happen to the running time of the `enqueue` and `dequeue` methods if your queue only had a front pointer? Explain your reasoning.

`enqueue`: O(N)

`dequeue`: O(1)

Why?

For enqueue you would have to travel through your linked list to the end before you could add a new node, hence O(N). With the end pointer you could travel directly to the end of the list which is O(1).

`String`

- `.length()` Get the length of the `String`. $O(1)$.
- `.charAt(i)` Get the `char` at index `i`. $O(1)$.
- `.split(" ")` Split a string by spaces and store it in a string[].
- `.substring(i, j)` Get the substring between indices `i` and `j`. Index `i` is *inclusive*, and index `j` is *exclusive*. $O(1)$. For example:

```
String x = "abcdefg";
String y = x.substring(2, 4);
// y now has the value "cd"
```

`ArrayList<T>  // Where T is a type, like String or Integer`

- `.add(i, X)` Add element `X` to the list at index `i`. If no `i` is provided, add an element to the end of the list. Adding to the end runs in $O(1)$.
- `.get(i)` Get the element at position `i`. Runs in $O(1)$.
- `.set(i, X)` Set the element at position `i` to the value `X`. $O(1)$.
- `.size()` Get the number of elements. $O(1)$.

`HashSet<T>  // Where T is a type, like String or Integer`

- `.size()` Compute the size. $O(1)$.
- `.add(X)` Add the value `X` to the set. If it's already in the set, do nothing. $O(1)$.
- `.contains(X)` Return a `boolean` indicating if `X` is in the set. $O(1)$.
- `.remove(X)` Remove `X` from the set. If `X` was not in the set, do nothing. $O(1)$.

`HashMap<K, V>  // Where K and V are the key and value types, respectively.`

- `.size()` Compute the size. $O(1)$.
- `.containsKey(X)` Determines if the map contains a value for the key `X`. To get that value, use `.get()`. $O(1)$.
- `.get(X)` Gets the value for the key `X`. If `X` is not in the map, return `null`. $O(1)$.
- `.put(k, v)` Map the key `k` to the value `v`. If there was already a value for `k`, replace it. $O(1)$.
- `.keySet()` Return a `Set` containing the keys in the map. Useful for iterating over. $O(1)$.

To iterate over a `HashSet<T>`, use

```
for (T v : nameOfSet) {
  // v is the current element of the set.
}
```

This can be combined with `HashMap`'s `.keySet()` to iterate over a `HashMap`.